```
/*
 * Library for reading settings from a configuration file stored in a SD
 * Based in SDConfigFile by Bradford Needham
(https://github.com/bneedhamia/sdconfigfile)
 * Licensed under LGPL version 2.1
 * a version of which should have been supplied with this file.
 *
 * The library supports one #define:
 *   #define SDCONFIG_DEBUG 1 // to print file error messages.
 */

#ifndef ConfigFile_h
#define ConfigFile_h

#include <Arduino.h>
#include <SdFat.h>

//#define CONFIGFILE_DEBUG 1

template <uint8_t K>
class ConfigFile {
  private:
    File _file;             // the open configuration file
    boolean _atEnd;         // If true, there is no more of the file to read.
    char _line[K];            // the current line of the file (see _lineLength)
                            // Allocated by begin().
    //uint8_t _lineSize;      // size (bytes) of _line[]
    uint8_t _lineLength;    // length (bytes) of the current line so far.
    uint8_t _valueIndex;      // position in _line[] where the value starts
                            //  (or -1 if none)
                            // (the name part is at &_line[0])

  public:
    boolean begin(const char *configFileName);
    void end();
    boolean readNextSetting();
    boolean nameIs(const char *name);
    const char *getName();
    //const char *getValue();
    int getIntValue();
    //IPAddress getIPAddress();
    //boolean getBooleanValue();
    //char *copyValue();
};

/*
 * Opens the given file on the SD card.
 * Returns true if successful, false if not.
 *
 * configFileName = the name of the configuration file on the SD card.
 *
 * NOTE: SD.begin() must be called before calling our begin().
 */
template <uint8_t K> boolean ConfigFile<K>::begin(const char *configFileName) {
  _lineLength = 0;
  //_lineSize = 0;
  _valueIndex = -1;
  _atEnd = true;

  /*
   * Allocate a buffer for the current line.
```

```
   */
  //_lineSize = K + 1;
  //_line = (char *) malloc(_lineSize);
  if (_line == 0) {
#ifdef CONFIGFILE_DEBUG
    Serial.println("out of memory");
#endif
    _atEnd = true;
    return false;
  }

  /*
   * To avoid stale references to configFileName
   * we don't save it. To minimize memory use, we don't copy it.
   */

  //_file = SD.open(configFileName, FILE_READ);
  if (!_file.open(configFileName,O_RDONLY)) {
#ifdef CONFIGFILE_DEBUG
    Serial.print("Could not open SD file: ");
    Serial.println(configFileName);
#endif
    _atEnd = true;
    return false;
  }

  // Initialize our reader
  _atEnd = false;

  return true;
}

/*
 * Cleans up our ConfigFile File object.
 */
template <uint8_t K> void ConfigFile<K>::end() {
  if (_file) {
    _file.close();
  }
  _atEnd = true;
}

/*
 * Reads the next name=value setting from the file.
 * Returns true if the setting was successfully read,
 * false if an error occurred or end-of-file occurred.
 */
template <uint8_t K> boolean ConfigFile<K>::readNextSetting() {
  int bint;

  if (_atEnd) {
    return false;  // already at end of file (or error).
  }

  _lineLength = 0;
  _valueIndex = -1;

  /*
   * Assume beginning of line.
   * Skip blank and comment lines
   * until we read the first character of the key
```

```
   * or get to the end of file.
   */
  while (true) {
    bint = _file.read();
    if (bint < 0) {
      _atEnd = true;
      return false;
    }

    if ((char) bint == '#') {
      // Comment line.  Read until end of line or end of file.
      while (true) {
        bint = _file.read();
        if (bint < 0) {
          _atEnd = true;
          return false;
        }
        if ((char) bint == '\r' || (char) bint == '\n') {
          break;
        }
      }
      continue; // look for the next line.
    }

    // Ignore line ends and blank text
    if ((char) bint == '\r' || (char) bint == '\n'
        || (char) bint == ' ' || (char) bint == '\t') {
      continue;
    }

    break; // bint contains the first character of the name
  }

  // Copy from this first character to the end of the line.

  while (bint >= 0 && (char) bint != '\r' && (char) bint != '\n') {
    if (_lineLength >= K) { // -1 for a terminating null.
      _line[_lineLength] = '\0';
#ifdef CONFIGFILE_DEBUG
      Serial.print("Line too long: ");
      Serial.println(_line);
#endif
      _atEnd = true;
      return false;
    }

    if ((char) bint == '=') {
      // End of Name; the next character starts the value.
      _line[_lineLength++] = '\0';
      _valueIndex = _lineLength;

    } else {
      _line[_lineLength++] = (char) bint;
    }

    bint = _file.read();
  }

  if (bint < 0) {
    _atEnd = true;
    // Don't exit. This is a normal situation:
```

```
    // the last line doesn't end in newline.
  }
  _line[_lineLength] = '\0';

  /*
   * Sanity checks of the line:
   *   No =
   *   No name
   * It's OK to have a null value (nothing after the '=')
   */
  if (_valueIndex < 0) {
#ifdef CONFIGFILE_DEBUG
    Serial.print("Missing '=' in line: ");
    Serial.println(_line);
#endif
    _atEnd = true;
    return false;
  }
  if (_valueIndex == 1) {
#ifdef CONFIGFILE_DEBUG
    Serial.print("Missing Name in line: =");
    Serial.println(_line[_valueIndex]);
#endif
    _atEnd = true;
    return false;
  }

  // Name starts at _line[0]; Value starts at _line[_valueIndex].
  return true;

}

/*
 * Returns true if the most-recently-read setting name
 * matches the given name, false otherwise.
 */
template <uint8_t K> boolean ConfigFile<K>::nameIs(const char *name) {
  if (strcmp(name, _line) == 0) {
    return true;
  }
  return false;
}

/*
 * Returns the name part of the most-recently-read setting.
 * or null if an error occurred.
 * WARNING: calling this when an error has occurred can crash your sketch.
 */
template <uint8_t K> const char *ConfigFile<K>::getName() {
  if (_lineLength <= 0 || _valueIndex <= 1) {
    return 0;
  }
  return &_line[0];
}

/*
 * Returns the value part of the most-recently-read setting,
 * or null if there was an error.
 * WARNING: calling this when an error has occurred can crash your sketch.
 */
/*
```

```
244 template <uint8_t K> const char *ConfigFile<K>::getValue() {
245   if (_lineLength <= 0 || _valueIndex <= 1) {
246     return 0;
247   }
248   return &_line[_valueIndex];
249 }
250 */
251 /*
252  * Returns a persistent, dynamically-allocated copy of the value part
253  * of the most-recently-read setting, or null if a failure occurred.
254  *
255  * Unlike getValue(), the return value of this function
256  * persists after readNextSetting() is called or end() is called.
257  */
258 /*
259 template <uint8_t K> char *ConfigFile<K>::copyValue() {
260   char *result = 0;
261   int length;
262 
263   if (_lineLength <= 0 || _valueIndex <= 1) {
264     return 0; // begin() wasn't called, or failed.
265   }
266 
267   length = strlen(&_line[_valueIndex]);
268   result = (char *) malloc(length + 1);
269   if (result == 0) {
270     return 0; // out of memory
271   }
272 
273   strcpy(result, &_line[_valueIndex]);
274 
275   return result;
276 }
277 */
278 /*
279  * Returns the value part of the most-recently-read setting
280  * as an integer, or 0 if an error occurred.
281  */
282 template <uint8_t K> int ConfigFile<K>::getIntValue() {
283   /*
284   const char str[5] = getValue();
285   if (!str) {
286     return 0;
287   }
288   return atoi(str);
289   */
290   if (_lineLength <= 0 || _valueIndex <= 1) return 0;
291   else return atoi(&_line[_valueIndex]);
292 }
293 
294 /*
295 IPAddress ConfigFile::getIPAddress(){
296   IPAddress ip(0,0,0,0);
297   const char *str = getValue();
298   int len = strlen(str);
299   char ipStr[len+1];
300   strncpy(ipStr,str,len); //char * strcpy ( char * destination, const char * source
); It is necessary to make a copy
301   ipStr[len] = '\0';
302   int i=0; int tmp;
303   const char *token = strtok(ipStr, ".");
```

```
  while (token != NULL ) {
    tmp = atoi(token);
    if(tmp < 0 || tmp > 255 || i > 3){
      ip={0,0,0,0};
      return ip; //IP does not have more than four octets and its values are smaller than 256
    }
    ip[i++] = (byte) tmp;
    token = strtok(NULL, ".");
  }
  return ip;
}
*/

/*
 * Returns the value part of the most-recently-read setting
 * as a boolean.
 * The value "true" corresponds to true;
 * all other values correspond to false.
 */
/*
template <uint8_t K> boolean ConfigFile<K>::getBooleanValue() {
  if (strcmp("true", getValue()) == 0) {
    return true;
  }
  return false;
}
*/
#endif

```